Haskell — Apontamentos

(Versão 1.16)

Pedro Quaresma de Almeida[1]

25 de Fevereiro de 2011

[1]Departamento de Matemática da Universidade de Coimbra.

Conteúdo

Capítulo 1

Introdução

A linguagem *Haskell*[1] (Bird, 1998; Hammond *et al.*, 1997; Hudak, 2000; Hudak *et al.*, 1997; Hutton, 2007; Peterson *et al.*, 1997; Thompson, 1996) é uma linguagem funcional[2] (pura) com tipos de dados bem definidos (Bird, 1998), ou seja, os modelos para as estruturas de informação são baseados na Teoria dos Conjuntos, e as operações são descritas através de funções entre conjuntos.

Os programas mais divulgados para se trabalhar com a linguagem Haskell são o interpretador *Hugs*[3] (Jones & Peterson, 1997) e o compilador e interpretador *GHC – the Glasgow Haskell Compiler*[4] sendo que este último providencia também um interpretador, o *GHCI*, com um modo de utilização idêntica ao *Hugs*.

A diferença mais visível entre um compilador e um interpretador é dada pela interacção com o utilizador. No caso de um compilador é necessário utilizar um editor de texto para escrever um programa que é depois transformado pelo compilador de uma forma automática num executável. No caso do interpretador não só é possível escrever programas e avaliá-los através do próprio interpretador (no caso do *Hugs* e do *GHCI* só é possível calcular o valor de expressões), como no caso de se utilizar um editor externo para a escrita do programa é depois necessário ter o interpretador em memória de forma a executar o programa. Os interpretadores permitem uma testagem dos programas de uma forma interactiva o que no caso de uma linguagem funcional (funções independentes umas das outras) é bastante interessante.

De seguida vai-se centrar a atenção na utilização dos interpretadores pois será esta a forma mais usual de desenvolver programas em Haskell.

1.1 Utilização dos Interpretadores

Ao invocar-se o interpretador este carrega de imediato o módulo `Prelude.hs`, que constitui o módulo básico da linguagem *Haskell*.

Toda a interacção com o programa *Hugs*, ou *GHCi* (The GHC Team, n.d.) desenvolve-se no âmbito de um dado módulo, tal facto é posto em evidência na linha de comando do programa:

```
Prelude>
```

[1] `http://www.haskell.org/`
[2] `http://www.mat.uc.pt/~pedro/cientificos/Funcional.html`
[3] `http://www.haskell.org/hugs/`
[4] `http://www.haskell.org/ghc/`

Se se invocar o interpretador conjuntamente com o nome de um ficheiro, já previamente construído, e que contenha um módulo, por exemplo `Fact.hs`, contendo o módulo `Fact`, a linha de comando do interpretador passará a ser:

```
Fact>
```

A interacção com o interpretador limitar-se-á a:

1. Avaliação de expressões. As expressões têm de estar contidas numa só linha. Por exemplo:

   ```
   Prelude> 6*4
   24
   Prelude>
   ```

2. Avaliação de comandos. Entre estes temos: "`:q`", para abandonar o interpretador; "`:?`", para obter uma lista dos comandos disponíveis; e "`:load` <nome_de_ficheiro>" para carregar um novo módulo. Um comando interessante numa fase de aprendizagem é o comando `set +t`. Este comando `set`, com o argumento `+t`, modifica o comportamento do interpretador de molde a que este passe a mostrar a informação acerca do tipo dos resultados dos cálculos da expressões. Por exemplo

   ```
   Prelude> :set +t
   Prelude> 6*3.0
   18.0 :: Double
   Prelude>
   ```

1.2 Utilização do Compilador `ghc`

A utilização do compilado `ghc` (The GHC Team, n.d.) requer que se defina o módulo `Main` e, dentro deste, a função zero-ária `main`. Na construção deste modo aplicam-se todas as regras que veremos à frente para a construção de módulos em Haskell, nomeadamente a importação de outros módulos.

Vejamos como é que poderiamos construir então um executável que nos desse o valor da função de Ackermann para os valores de entrada 3 e 9.

Primeiro definia-se um módulo (ficheiro) para a função de Ackermann (veremos os pormenores de uma tal definição mais à frente.

```
module Ackermann(ackermann) where
 ackermann :: (Integer,Integer) → Integer
 ackermann (0,y) = y+1
 ackermann (x,0) = ackermann(x-1,1)
 ackermann (x,y) = ackermann(x-1,ackermann(x,y-1))
```

De seguida constroí-se o módulo `Main` que mais não faz do que importar o módulo definido acima, definir um ponto de entrada o qual deve ter como resultado a escrita do valor que se pretende mostrar.

```
module Main where
 import Ackermann

 main=print (ackermann (3,7))
```

Finalmente é só uma questão de usar o compilador `ghc` para produzir o executável.

```
ghc --make  -o Main Main.hs
```

a opção `--make` tem o efeito de chamar um modo *Makefile* próprio do compilador `ghc` o qual automatiza todos os passos necessários à construção do programa executável, nomeadamente a compilação de todos os diferentes módulos que, de alguma forma, estão relacionados com o módulo `Main`.

1.3 Escrita de Programas em Haskell

Para a escrita de programas, isto é funções, é necessário usar um editor exterior. As novas definições podem ser incorporadas na definição de um novo módulo *Haskell*, ou então escritas de uma forma solta, caso em que serão automaticamente incorporadas no módulo `Main`. Posteriormente as novas definições podem ser carregado no interpretador de *Haskell* através do comando `:load` <*nome_do_ficheiro*>.

Comentários Em *Haskell* os comentários são introduzidos pela sequência "`--`", à direita da qual todo o restante conteúdo da linha é ignorado.

Podemos ter comentários que se extendem por várias linhas bastando para tal colocar a sequência referida no princípio de cada linha.

Para poder colocar toda uma secção de texto como comentário tem-se ainda a possiblidade de definir um bloco de comentários através das sequências {`-`, `-`}, abrir e fechar bloco respectivamente. Podemos definir vários blocos de comentários aninhados uns nos outros.

Indentação É importante notar que a forma como as várias definições são escritas em *Haskell* não é "inocente", isto é, em *Haskell* a indentação é significativa, sendo que a forma como as definições estão escritas afecta a avaliação das mesmas.

Ao contrário de muitas outras linguagens (*Pascal*, *C*, *Fortran*, ...) em que a indentação, embora aconselhada, não é significativa, sendo somente uma forma de organizar o texto para permitir uma leitura mais fácil, em *Haskell* a indentação é significativa. As regras precisas são facilmente assimiláveis pois que se tratam das regras que são aconselhadas para a escrita de programas nas outras linguagens. Temos então:

1. Se a linha começa mais à frente do que começou a linha anterior, então ela deve ser considerada uma continuação da linha anterior. Assim por exemplo, escrever

   ```
   quadrado x
     = x*x
   ```

   é o mesmo que escrever

   ```
   quadrado x = x*x
   ```

2. Se uma linha começa na mesma coluna que a anterior, então elas são consideradas definições independentes.

3. Se uma linha começa mais atrás do que a anterior, então essa linha não pertence à mesma lista de definições.

Se quisermos separar, de forma explícita, duas definições podemos usar o separador "`;`". Por exemplo:

```
dobro x = 2*x ; triplo x = 3*x
```

1.3.1 A utilização do *(X)emacs*

O desenvolvimento de programas em *Haskell* é muito facilitado caso se use o editor *(X)emacs*, caso em que se pode usar o *haskell-mode*[5], um modo de edição do *(X)emacs* especializado para o desenvolvimento de programas em *Haskell*. De seguida transcreve-se a informação (`C-h m`) existente para este modo.

```
Haskell mode:
Major mode for editing Haskell programs.  Last adapted for Haskell 1.4.
Blank lines separate paragraphs, comments start with '-- '.

M-; will place a comment at an appropriate place on the current line.
C-c C-c comments (or with prefix arg, uncomments) each line in the region.

Literate scripts are supported via 'literate-haskell-mode'.  The
variable 'haskell-literate' indicates the style of the script in the
current buffer.  See the documentation on this variable for more
details.

Modules can hook in via 'haskell-mode-hook'.  The following modules
are supported with an 'autoload' command:

   'haskell-font-lock', Graeme E Moss and Tommy Thorn
     Fontifies standard Haskell keywords, symbols, functions, etc.

   'haskell-decl-scan', Graeme E Moss
     Scans top-level declarations, and places them in a menu.

   'haskell-doc', Hans-Wolfgang Loidl
     Echoes types of functions or syntax of keywords when the cursor is idle.

   'haskell-indent', Guy Lapalme
     Intelligent semi-automatic indentation.

   'haskell-simple-indent', Graeme E Moss and Heribert Schuetz
     Simple indentation.

   'haskell-hugs', Guy Lapalme
     Interaction with Hugs interpreter.

Module X is activated using the command 'turn-on-X'.  For example,
'haskell-font-lock' is activated using 'turn-on-haskell-font-lock'.
For more information on a module, see the help for its 'turn-on-X'
function.  Some modules can be deactivated using 'turn-off-X'.  (Note
that 'haskell-doc' is irregular in using 'turn-(on/off)-haskell-doc-mode'.)
```

[5] `http://www.haskell.org/haskell-mode/`

```
Use ‘haskell-version’ to find out what version this is.

Invokes ‘haskell-mode-hook’ if not nil.

----

Font-Lock minor mode (indicator Font):
Toggle Font Lock Mode.
With arg, turn font-lock mode on if and only if arg is positive.

When Font Lock mode is enabled, text is fontified as you type it:

 - Comments are displayed in ‘font-lock-comment-face’;
 - Strings are displayed in ‘font-lock-string-face’;
 - Documentation strings (in Lisp-like languages) are displayed in
   ‘font-lock-doc-string-face’;
 - Language keywords ("reserved words") are displayed in
   ‘font-lock-keyword-face’;
 - Function names in their defining form are displayed in
   ‘font-lock-function-name-face’;
 - Variable names in their defining form are displayed in
   ‘font-lock-variable-name-face’;
 - Type names are displayed in ‘font-lock-type-face’;
 - References appearing in help files and the like are displayed
   in ‘font-lock-reference-face’;
 - Preprocessor declarations are displayed in
  ‘font-lock-preprocessor-face’;

   and

 - Certain other expressions are displayed in other faces according
   to the value of the variable ‘font-lock-keywords’.

Where modes support different levels of fontification, you can use the variable
‘font-lock-maximum-decoration’ to specify which level you generally prefer.
When you turn Font Lock mode on/off the buffer is fontified/defontified, though
fontification occurs only if the buffer is less than ‘font-lock-maximum-size’.
To fontify a buffer without turning on Font Lock mode, and regardless of buffer
size, you can use M-x font-lock-fontify-buffer.

See the variable ‘font-lock-keywords’ for customization.

----

Column-Number minor mode (indicator):
Toggle Column Number mode.
With arg, turn Column Number mode on iff arg is positive.
```

```
When Column Number mode is enabled, the column number appears
in the mode line.

----

Line-Number minor mode (indicator):
Toggle Line Number mode.
With arg, turn Line Number mode on iff arg is positive.
When Line Number mode is enabled, the line number appears
in the mode line.

----

Haskell-Indent minor mode (indicator Ind):
``intelligent'' Haskell indentation mode that deals with
the layout rule of Haskell.  tab starts the cycle
which proposes new possibilities as long as the TAB key is pressed.
Any other key or mouse click terminates the cycle and is interpreted
except for RET which merely exits the cycle.
Other special keys are:
    C-c = inserts an =
    C-c | inserts an |
    C-c o inserts an | otherwise =
these functions also align the guards and rhs of the current definition
    C-c w inserts a where keyword
    C-c . aligns the guards and rhs of the region.
    C-c > makes the region a piece of literate code in a literate script

Note: M-C-\ which applies tab for each line of the region also works
but it stops and ask for any line having more than one possible indentation.
Use TAB to cycle until the right indentation is found and then RET to go the
next line to indent.

Invokes `haskell-indent-hook' if not nil.

----

Haskell-Doc minor mode (indicator Doc):
Enter haskell-doc-mode for showing fct types in the echo area (see variable docstring).
```

Antes de tratar da escrita de módulos em *Haskell*, vejamos quais são os tipos de dados disponíveis no módulo `Prelude` e como escrever expressões nesse módulo.

Capítulo 2

Tipos de Dados

O *Haskell* é uma linguagem de programação com uma disciplina de tipos rigorosa ("strongly typed"), quer isto dizer que toda a entidade num programa em *Haskell* tem um, e um só, tipo, sendo sempre possível determinar o tipo de uma determinada entidade. O contra-ponto deste tipo de linguagem são as linguagens sem tipos, ou melhor linguagens em que todas as entidades partilham um mesmo tipo, um exemplo deste tipo de linguagens é a linguagem Lisp. Existem argumentos a favor e contra cada uma destas duas aproximações, uma das grandes vantagens das linguagens com uma disciplina de tipos rigorosa reside na possibilidade de constatar a correcção dos programas através da verificação dos tipos, as linguagens deste tipo são também as que melhor suportam a modularidade e a abstracção.

Em termos práticos temos que um dado elemento tem sempre um tipo bem definido, de igual modo toda e qualquer função tem um domínio e um co-domínio bem definidos.

Em *Haskell* temos então um conjunto de tipos de base, pré-definidos, e um conjunto de construtores que permitem ao programador criar novos tipos a partir dos tipos de base.

2.1 Tipos de Base

O *Haskell* possui os seguintes tipos de base:

- Tipo Unitário ($\equiv \mathbf{1}$);
- Tipo Lógico, `Bool`;
- Tipo Caracter, `Char`;
- Tipos Numéricos:
  - `Int`;
  - `Integer`;
  - `Float`;
  - `Double`;

Existe ainda o tipo Vazio, `Void`, que vai ser importante aquando da definição de funções parciais. O único elemento deste tipo, $\perp$, é interpretado como o valor *indefinido*,

permitindo deste modo a definição de funções que são indefinidas para certa classe de argumentos. É de notar que, em consequência do que se acabou de dizer, o elemento $\perp$ vai pertencer a todos os tipos.

Vejamos cada um destes tipos mais em pormenor:

Convenção Sintáctica: Os identificadores dos elementos começam por uma letra minúscula, os identificadores de tipos, e os identificadores dos construtores começam por uma letra maiúscula.

2.1.1 Tipo Unitário

O tipo unitário, `Unit`, é uma implementação do conjunto **1**.

Tipo	`()`
Valores	`()`

2.1.2 Bool

Valores Lógicos e símbolos proposicionais para uma lógica proposicional bivalente.

Tipo	`Bool`
Valores	`True`; `False`; `otherwise` ($\doteq$ True)
Operadores	`&&` (conjunção); $\|\|$ (disjunção); `not` (negação)
Predicados	$==, /=, <, <=, >=, >$

Os identificadores `True`, e `False` correspondem aos dois construtores 0-ários do conjunto `Bool`. O identificador `otherwise`, cujo valor é igual a `True` vai ser útil aquando da definição de funções com ramos.

2.1.3 Char

Caracteres, símbolos do alfabeto "Unicode".

Tipo	`Char`
Valores	símbolos da *Tabela Unicode* entre plicas, por exemplo `'a'`
Operadores	

A exemplo de outras linguagens o *Haskell* possui um conjunto de caracteres especiais: '`\t`', espaço tabular (tab); '`\n`', mudança de linha (newline); '`\\`' barra inclinada para trás, (backslash); '`\'`', plica; '`\"`', aspas.

Tem-se acesso às duas funções de transferência usuais entre valores do tipo `Char` e valores do tipo `Int`, estas funções designam-se por: `ord :: Char -> Int` e `chr :: Int -> Char`.

Além destas funções temos também acesso às funções de transferência definidas para todo e qualquer tipo enumerado. No caso do tipo `Char` estas funções têm valores inteiros entre 0 e $2^{16} - 1$ inclusivé:

`toEnum` (Int $\rightarrow$ Char), por exemplo `(toEnum 97)::Char = 'a'`;

`fromEnum` (Char $\rightarrow$ Int), por exemplo `fromEnum 'a' = 97`.

Na expressão `(toEnum 97)::Char = 'a'`, a componente `::Char` serve para retirar a ambuiguidade que existe quanto ao co-domínio da função de transferência `toEnum`, isto dado que o co-domínio é dado pela classe de todos os tipos enumeráveis, mais à frente precisar-se-á o que se entende por classe de tipos. Esta função é um exemplo de uma função polimórfica (veremos isso mais à frente) sendo por tal possível aplica-la em diferentes situações.

2.1.4 Int, Integer, Float, Double

Valores numéricos, inteiros (`Int, Integer`), e reais (`Float, Double`).

Int, Integer

Inteiros.

Tipos	`Int, Integer`
Valores	`Int`, inteiros de comprimento fixo. A gama é de, pelo menos, -2^{29} a $2^{29}-1$. `Integer`, inteiros de comprimento arbitrário.
Operadores	`+, *, -, negate` ($\doteq$ `-` unário), `quot, div, rem, mod`

A gama de variação do tipo `Int` é nos dada pelas constantes `primMinInt`, e `primMaxInt`.

O operador `quot` faz o arredondamento "em direcção ao $-\infty$". O operador `div` faz o arredondamento "em direcção ao 0". Estes operadores verificam as seguintes equações:

```
(x `quot` y)*y+(x `rem` y) == x
(x `div` y)*y+(x `mod` y)  == x
```

A escrita de uma função binária entre plicas (acento grave), por exemplo `` `quot` `` permite a sua utilização como um operador infixo.

Funções de transferência: `even` (par), e `odd` (impar).

Float, Double

Reais.

Tipos	`Float, Double`
Valores	`Float`, reais precisão simples. `Double`, reais precisão dupla.
Operadores	`+, *, /, -, negate, ^, pi, exp, log, sqrt, **, logBase, sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, asinh, acosh, atanh.`

Em que `x^n` dá-nos a potência inteira de x, `x**y` dá-nos a exponencial de base x de y, e `logBase x y` dá-nos o logaritmo de base x de y.

Comum a todos os tipos numéricos temos os operadores `abs` e `signum`. Em que `signum(x)` é igual a -1, 0 ou 1, consoante x seja negativo, nulo ou positivo, respectivamente.

Funções de transferência: `ceiling, floor, truncate, round`. A função `ceiling(x)` dá-nos o menor inteiro que contêm x, `floor(x)` dá-nos o maior inteiro contido em x.

2.1.5 Tipo Vazio

Tipo	`Void`
Valores	$\perp$

Dado que o *Haskell* é uma linguagem não-estrita (veremos mais à frente o significado preciso desta afirmação) o valor $\perp$ (indefinido) está presente em todos os tipos.

2.2 Tipos Estruturados

O *Haskell* possui um conjunto de construtores de tipos que possibilitam a construção de novos tipos a partir dos tipos de base. Temos assim a possibilidade de construir novos tipos a partir do co-produto, do produto, e da exponenciação de conjuntos.

2.2.1 Produtos, (N-uplos)

O tipo correspondente a um dado n-uplo implementa o produto cartesiano dos tipos que compõem o n-uplo.

$$T = T_1 \times T_2 \times \cdots \times T_m$$

Tipo	`(`$\mathtt{T}_1$`,...,`$\mathtt{T}_n$`)`
Valores	`(`$\mathtt{e}_1$`,...,`$\mathtt{e}_k$`)`, com $k \geq 2$
Construtores	`(,...,)`
Operadores	`fst, snd`, só aplicáveis a pares.

Os operadores `fst(x,y) = x`, e `snd(x,y)=y` dão-nos as duas projecções próprias do produto cartesiano.

2.2.2 Co-produto (Enumeração)

O *Haskell* possuí também tipos definidos por enumeração, podemos por exemplo considerar que os tipos simples podem ser definidos desse modo.

O tipo co-produto implementa o co-produto de conjuntos, ou dito de outra forma, a união disjunta de conjuntos.

$$T = T_1 + T_2 + \cdots + T_m$$

Tipo	$\mathtt{T}_1 \mid \ldots \mid \mathtt{T}_n$. Com $\mathtt{T}_i$ um tipo, ou um construtor unário.
Valores	`e`, com `e` um elemento de um dos tipos que definem o co-produto.
Construtores	os construtores dos diferentes tipos, assim como construtores unários definidos caso a caso.

A definição de tipos por enumeração, e a utilização dos contructores próprios dos co-produtos irá ser explorada mais à frente aquando do estudo da definição de novos tipos em *Haskell*.

2.2.3 Listas

O conjunto das sequências finitas, eventualmente vazias, de elementos de um dado tipo A, isto é, $A^* = A^0+A^1+A^2+\cdots+A^n$, é implementado através do tipo listas homogéneas.

Para um dado tipo de base `A` temos:

Lista `A` = Vazia+`A`×Lista `A`

Tipo	`[A]`, listas de elementos de um tipo `A`
Valores	`[]` (lista vazia); `[`e_0`,...,`e_m`]` (lista não vazia)
Construtores	`[]` : 1 $\longrightarrow$ Lista $*$ $\longmapsto$ `[]` : : A × Lista A $\longrightarrow$ Lista A (a,l) $\longmapsto$ l′ = a:l `[`e_0`,`e_1`,...,`e_m`]` $\doteq$ e_0`:(`e_1`:(...:(`e_m `: []))...)`
Operadores	`++, head, last, tail, init, nul, length, !!, foldl, foldl1, scanl, scanl1, foldr, foldr1, scanr, scanr1, iterate, repeat, replicate, cycle, take, drop, splitAt, takeWhile, dropWhile, span, break, lines, words, mlines, mwords, reverse, and, or, any, all, elem, notElem, lookup, sum, product, maximun, minimun, concatMap, zip, zip3, zipWith, zipWith3, unzip, unzip3`

De entre de todos estes operadores vejamos a definição de `last` e de `init`:

`last(`[e_0`,`e_1`,...,`e_m`]) =` e_m

`init(`[e_0`,`e_1`,...,`e_m`]) =` [e_0`,`e_1`,...,`e_{m-1}`]`

O operador `!!` permite usar uma lista como uma tabela uni-dimensional com índices a começar no zero. Por exemplo:

`[`e_0`,`e_1`,...,`e_m`] !!` `k =` e_k, $0 \leq k \leq m$

O operador `++` dá-nos a concatenação de duas listas. Por exemplo:

`[1,2,3]++[4,5,6] = [1,2,3,4,5,6]`

O *Haskell* possuí duas formas de construção de listas sem que para tal seja necessário escrever a lista com todos os seus elementos. Vejamos de seguida esses dois mecanismos.

Definição de uma lista através de uma gama de variação. Podemos definir uma lista de elemento numéricos através dos seus limites, ou ainda através dos seus limites e do incremento entre valores. Vejamos esses dois casos:

- `[n..m]` é a lista `[n,n+1,n+2,...,m]`. Se o valor de `n` exceder o de `m`, então a lista é vazia. Por exemplo:

```
[2..5] = [2,3,4,5]
[3.1..7.0] = [3.1,4.1,5.1,6.1]
[7..3] = []
```

- `[n,p..m]` é a lista de todos os valores desde `n` até `m`, em passos de `p-n`. Por exemplo:

```
[7,6..3] = [7,6,5,4,3]
[0.0,0.4..1.0] = [0.0,0.4,0.8]
```

Em ambos os caso o último elemento da lista é o último elemento a verificar a condição requerida, sem que o seu valor ultrapasse o valor limite.

Definição de uma lista por compreensão. O *Haskell* permite a construção de lista através de uma definição por compreensão, à imagem daquilo que estamos habituados em relação à definição de conjuntos. Pegando nessa analogia vejamos como se procede à definição de listas por compreensão.

Para conjuntos temos:

$$\{E(x) \mid x \in C \wedge P_1(x) \wedge \ldots \wedge P_n(x)\}$$

com $E(x)$ uma expressão em x, C um conjunto que é suposto conter x, e os vários $P_i(x)$ são proposições em x.

Para listas em *Haskell* tem-se:

```
[E(x) | x <- l, P1(x), ..., Pn(x)]
```

com `l` uma lista.

Por exemplo: `[2*a | a <- [2,4,7], even a, a>3]` dá-nos a lista de todos os `2*a` em que `a` pertence à lista `[2,4,7]`, e `a` é par, e `a` é maior do que 3, ou seja a lista `[8]`.

As sequências de proposições $P_i(x)$ pode ser vazia, por exemplo: `[2*a | a <- l]` é uma forma expedita de obter uma lista cujos elementos são em valor o dobro dos elementos de uma dada lista `l`.

É também possível definir, por compreensão, listas de tipos compostos, por exemplo listas de pares de elementos.

Um possível exemplo de uma definição deste tipo é-nos dada por:

```
[(x,y) | x <- ['a'..'c'], y <- [0..2]]
```

qual será a lista gerada por uma tal definição?

2.2.4 Sequências de Caracteres ("Strings")

As sequências de caracteres são suficientemente importantes para terem um tratamento especial, no entanto elas podem sempre ser consideradas como uma lista de elementos do tipo `char`.

Tipo	String
Valores	sequências de caracteres entre aspas. Por exemplo `"abc"` ($\doteq$ ['a','b','c'])
Operadores	Os operadores das listas.

2.2.5 Funções

As funções vão estar subjacentes a todas as manipulações da linguagem e implementam a exponenciação de conjuntos. Ou seja o conjunto de todas as funções de A para B é dado por B^A.

Tipo	`T1 -> T2`
Valores	`id`, `const`, ...
Construtores	`\ x -> exp(x)`
Operadores	$\cdot$, `curry`, `uncurry`, ...

O operador "$\cdot$" dá-nos a composição de funções. Os operadores `curry` e `uncurry` verificam, `curry f x y = f (x,y)` e `uncurry f p = f (fst p) (snd p)`.

A abstracção lambda permite-nos definir uma função sem lhe dar um nome, o que não nos impede a sua aplicação a um dado argumento. Por exemplo `(\x -> x^2) 4` daria o resultado `16`.

Embora a definição de funções seja habitualmente feita de forma diferente (mais à frente veremos como), pode-se atribuir um nome a uma função definida através da abstracção lambda, por exemplo `quadrado = (\x -> x^2)`

2.3 Expressões

A linguagem *Haskell* é uma linguagem com uma disciplina de tipos rigorosa, quer isto dizer que qualquer expressão em *Haskell* tem um e um só tipo bem definido.

A este propósito é de referir uma série de comandos que se revelam interessantes; um deles é o comando `:type` (ou `:t`). Este comando dá-nos, para uma determinada expressão, a informação acerca do tipo da expressão. Por exemplo:

```
Prelude> :t 'a'
'a' :: Char
```

A sintaxe <*valor*>`::`<*tipo*> tem o significado óbvio de <*valor*>$\in$<*tipo*>.

Outro comando útil, é o comando `:set +t`, ou melhor a aplicação do comando `:set` ao valor `+t`. Este comando modifica o comportamento do interpretador de modo a que este passa a explicitar (`+`) a informação do tipo (`t`) da expressão que é sujeita a avaliação. A situação inversa, valor por omissão, é conseguida através do comando `:set -t`. Por exemplo:

```
Prelude> 7+3
10
Prelude> :set +t
Prelude> 7+3
10 :: Int
```

A avaliação de uma expressão é feita por redução da expressão à sua forma normal. Podemos modificar o comportamento do interpretador de forma a que este explicite dados estatísticos (`:set +s`) acerca das reduções necessárias para a avaliação da expressão. Por exemplo

```
Prelude> :set +s
Prelude> 7+3
10 :: Int
(10 reductions, 10 cells)
```

Posto isto a escrita de expressões e a sua avaliação segue as regras usuais. Por exemplo:

```
Prelude> "Haskell"
"Haskell" :: String
Prelude> 5+3*10
35 :: Int
Prelude> head [3,56,79]
3 :: Int
Prelude> fst ('a',1)
'a' :: Char
Prelude> 4*1.0
4.0 :: Double
```

2.3.1 Precedência e Associatividade

A precedência dos vários operadores assim como o modo como se procede a sua associação é dada na seguinte tabela.

Precedência	associa à esquerda	associa à direita	não associativo
9	!!	.	—
8	—	^,^^,**	—
7	*,/,`div`,`mod`, `rem`,`quot`	—	—
6	+,-	—	—
5	—	:,++	\\
4	—	—	==,/=,<,<=,>,>=, `elem`,`notElem`
3	—	&&	—
2	—	||	—
1	>>,>>=	—	—
0	—	$,`seq`	—

Capítulo 3

Definição de Funções

O módulo `Prelude.hs` contêm um grande número de funções, entre elas temos as funções (operadores) referentes aos diferentes tipos de dados. Este módulo é imediatamente carregado quando se invoca o interpretador, sendo que esse facto é salientado no arranque e pela linha de comando.

```
...
Reading file "/usr/local/lib/hugs/lib/Prelude.hs":
...
Prelude>
```

Para definir novas funções é necessário utilizar um editor de texto para escrever as definições correspondentes num ficheiro, por exemplo `novas.hs` que posteriormente será carregado no interpretador através do comando `:load novas`.

Por exemplo, pode-se escrever um dado ficheiro de texto `novas.hs` contendo as seguintes linhas de texto:

```
quadrado x = x*x
dobro x = 2*x
```

depois torna-se necessário carregar estas novas definições no interpretador através do comando `:l novas`:

```
Prelude> :l novas
Reading file "novas.hs":

Hugs session for:
/usr/local/lib/hugs/lib/Prelude.hs
novas.hs
Main> dobro 3
6
Main> quadrado 3
9
Main>
```

A mudança da linha de comando reflecte o facto de o módulo `Main` ser o módulo em que, por omissão, são feitas as definições. Ou seja a menos que o nome de um novo módulo seja explicitamente definido o módulo em que são feitas as definições "soltas" é o módulo `Main`.

3.1 Definições Simples

A definição de uma função passa pela escrita de uma "equação" que determine o seu comportamento. A sintaxe é a seguinte:

$$< \textit{nome\_da\_função} > < \textit{argumento} > \texttt{=} < \textit{expressão} >$$

por exemplo

```
cubo x = x*x*x
```

A definição de funções deve ser entendida como uma regra de re-escrita da esquerda para a direita, a qual está implicitamente quantificada universalmente nas suas variáveis. Ou seja a leitura da definição de `cubo` é a seguinte:

$$\forall_{x \in T}\ \textit{cubo}\ x \rightarrow x \times x \times x$$

com T um dos tipos da linguagem *Haskell*, e $\rightarrow$ o operador de redução.

A avaliação de uma expressão que envolva a função `cubo` passa pela redução da expressão à sua forma normal, utilizando, entre outras, a regra de redução escrita acima. Por exemplo:

```
Main> cubo 3
27
```

teve como passos da redução $\textit{cubo}\ 3 \rightarrow (3 \times 3) \times 3 \rightarrow 9 \times 3 \rightarrow 27$.

Qual é o tipo da função `cubo`

```
Main> :t cubo
cubo :: Num a ⇒ a → a
```

A resposta aparentemente confusa tem a seguinte leitura, `cubo` é uma função de domínio A e co-domínio A (`a -> a`) em que A é um qualquer tipo numérico (`Num a =>`). Ou seja `cubo` é uma função polimórfica cujo tipo (`a->a`) pertence à classe dos tipos numéricos (`Num a`). A definição de uma instância concreta para a função `cubo` só acontece no momento da avaliação.

Um outro ponto que importa salientar é que as funções em *Haskell* têm um, e um só, argumento. Como proceder então para definir uma função a aplicar a dois argumentos?

Existem duas soluções possíveis: Em primeiro podemos agrupar os argumentos num n-uplo. Por exemplo:

```
soma1 :: (Int,Int) → Int
soma1 (x,y) = x + y
```

A outra solução é dada pela definição de uma função *curry*.

```
soma2 :: Int → (Int → Int)
soma2 x y = x + y
```

Estas duas definições são equivalentes dado serem a expressão da propriedade universal das funções. É de relembrar que a aplicação de funções associa à esquerda, ou seja `soma x y` deve ler-se como `(soma x) y`.

A vantagem da utilização de funções *curry* em detrimento das funções *não-curry* é dupla: por um lado utilizam-se argumentos não estruturados tornando deste modo a escrita mais simples; por outro lado ao aplicar-se uma função *curry* a um dos seus argumentos obtêm-se uma outra função que pode por sua vez ser um objecto interessante.

3.2 Definições Condicionais

A definição de uma função através de uma única equação não nos permite definir funções por ramos. Temos duas possibilidades para o fazer: através de expressões condicionais.

```
menor1 :: (Int,Int) → Int
menor1 (x,y) = if x ≤ y then x else y
```

ou então através de equações com "guardas".

```
menor2 :: (Int,Int) → Int
menor2 (x,y)
   | x ≤ y  = x
   | x > y  = y
```

As duas definições são equivalentes, no segundo caso as "guardas" são avaliados e a primeira a tomar o valor lógico de verdade determina o valor da função.

A segundo definição podia ser re-escrita do seguinte modo:

```
menor3 :: (Int,Int) → Int
menor3 (x,y)
   | x > y = y
   | otherwise = x
```

dado que o valor lógico de `otherwise` é *Verdade*, então este padrão determina o valor da função para todos os casos em que os outros ramos que o precedem não tenham o valor lógico de *Verdade*.

3.3 Definições Recursivas

A definição de funções recursivas em *Haskell* não só é possível, como a sua sintaxe é muito simples, e segue de perto a sintaxe matemática. Por exemplo a definição da função factorial é dada por:

```
fact :: Integer → Integer
fact n
   | n == 0 = 1
   | n >  0 = n*fact(n-1)
```

Na definição desta função levanta-se uma questão: e no caso em $n < 0$?

Para os valores de $n < 0$ a função factorial não está definida, ela é uma função parcial no conjunto dos inteiros, torna-se então necessário saber lidar com funções parciais. Em *Haskell* a forma de lidar com funções parciais passa por especificar situações de erro.

```
fact1 :: Integer → Integer
fact1 n
   | n < 0  = error "A função fact não está definida para n<0"
   | n == 0 = 1
   | n > 0  = fact1(n-1)*n
```

No entanto, e dado o tipo da função `fact1`, a função `error` tem de ser `String -> Integer`. O tipo da função `error` é na verdade `String -> A`, com `A` um tipo *Haskell*, ou seja `error` é uma função polimórfica cujo tipo do co-domínio depende da situação em que é aplicada.

O mesmo se passa com a função `‘*’`, o seu tipo é `A -> A -> A`, sendo depois instanciada para uma dada função concreta no momento da aplicação.

3.3.1 O uso de Acumuladores

Embora a definição recursiva de funções seja muito apelativa, o seu uso revela-se computacionalmente pesado. Vejamos um exemplo disso utilizando a definição acima escrita da função factorial, e vejamos o desenvolvimento do cálculo de `fact 4`.

```
fact 4
-> 4*(fact 3)
     -> 3*(fact 2)
          -> 2*(fact 1)
               -> 1*(fact 0)
                    -> 1
                    <- 1
               <- 1*1
          <- 2*1
     <- 3*2
<- 4*6
24
```

Temos então uma primeira fase em que há um simples desdobrar da definição de `fact`, e uma segunda fase em que são executados os cálculos. Este forma de proceder tem como consequência o criar de uma expressão que vai crescendo à medida que o valor do argumento cresce, sendo que os cálculos são feitos depois desta fase de desdobramento.

Uma outra forma de escrever a definição de `fact` faz uso de um acumulador, isto é, de uma variável auxiliar que serve para ir guardando os resultados parciais. Vejamos como proceder:

```
fact2 :: Int → Int
fact2 n
   | n < 0 = error "A função fact não está definida para n<0"
   | otherwise = factac 1 n

factac :: Int → Int → Int
factac ac n
   | n == 0 = ac
   | otherwise = factac ac*n (n-1)
```

Para o calcular `fact2 4` temos agora:

```
fact2 4
-> factac 1 4
   -> factac (1*4) 3
      -> factac (4*3) 2
         -> factac (12*2) 1
            -> factac (24*1) 0
            <- 24
         <- 24
      <- 24
   <- 24
<- 24
```

ou seja, na nova definição os cálculos são feitos imediatamente, não sendo necessário esperar pelo desdobrar de toda a definição.

Um exemplo extremos da diferença de eficiência entre os dois tipos de definição é dado pela implementação da função de *Fibonnacci*.

A definição matemática é nos dada pela seguinte expressão:

$$Fib(n) = \begin{cases} 1, & n = 0 \\ 1, & n = 1 \\ Fib(n-1) + Fib(n-2), & n \geq 2 \end{cases}$$

Uma possível implementação recursiva (clássica) desta função é dada por:

```
fib :: Int → Int
fib n
  | n == 0 = 1
  | n == 1 = 1
  | otherwise = fib(n-1) + fib(n-2)
```

Uma implementação recursiva, recorrendo a acumuladores é dada por:

```
fib1 :: Int → Int
fib1 n = fibac 1 1 n

fibac :: Int → Int → Int → Int
fibac ac1 ac2 n
  | n == 0 = ac2
  | n == 1 = ac2
  | otherwise = fibac ac2 (ac1+ac2) (n-1)
```

Para nos convencer-mo-nos da grande diferença de eficiência entre as duas definições, podemos calcular o valor de $Fib(35)$, através das duas implementações acima descritas, antes de o fazer vai-se modificar o comportamento do interpretador através do comando `:set +s`.

```
Fact> fib 35
14930352
(281498375 reductions, 404465770 cells, 4321 garbage collections)
Fact>
Fact> fib1 35
14930352
(358 reductions, 608 cells)
Fact>
```

Escusado será dizer que a diferença de tempo (real) entre os dois cálculos foi muito significativa.

3.4 Definições Polimórficas

A definição de funções polimórficas em *Haskell* é possível dado existirem variáveis de tipo, isto é variáveis cujo domínio de variação é o conjunto de todos os tipos em *Haskell*. Aquando do estudo das classes (capítulo 4) ver-se-á isto com mais profundidade.

Podemos então re-definir a função factorial como uma função polimórfica capaz de aceitar argumentos dos tipos `Int`, `Integer` e mesmo do tipo `Float`.

```
fact2 :: (Ord a, Num a) ⇒ a → a
fact2 n
   | n < 0  = error "A função fact não está definida para n<0"
   | n == 0 = 1
   | n > 0  = fact2(n-1)*n
```

Ou seja a função `fact2` é uma função polimórfica cujo tipo é `A -> A`, com o tipo `A` pertencente à classe dos tipos numéricos e com ordenação.

Outro exemplo

```
identidade :: a → a
identidade x = x
```

neste caso a função é aplicável a um qualquer tipo sem restrições, isto é pertence à classe mais genérica.

3.5 Associação de Padrões

Em alguns casos na definição de uma função por ramos, os diferentes casos a considerar são reduzidos. Por exemplo na definição da função de procura de um dado elemento numa lista. Existem dois casos diferentes a considerar: ou a lista está vazio, ou não. No último caso ou o elemento é igual à cabeça da lista, ou temos que considerar a procura no resto da lista. Temos então

```
procura :: (a,[a]) → Bool
procura (elem,l) = if l == [] then
                      False
                   else if elem == head(l) then
                           True
                        else
                           procura(elem,tail(l))
```

com `head` e `tail` duas funções pré-definidas no módulo `Prelude` com o significado óbvio. No entanto a definição de `procura` pode ficar mais clara e concisa se se usar os construtores próprios de uma lista de uma forma explícita.

```
procura1 (elem,[]) = False
procura1 (elem,hd:tl) = if elem == hd then
                           True
                        else
                           procura(elem,tl)
```

Nesta definição usou-se a técnica de *associação de padrões* ("pattern matching").

A associação de padrões está ligada aos tipos de dados definidos, por utilização dos seus construtores. No caso do tipo lista temos dois casos, padrões, a explorar: uma lista ou é a lista vazia; ou é uma lista constituída por um elemento (a cabeça da lista) e por uma lista (a cauda da lista). Os construtores definidos são `[]`, para a lista vazia, e `:` para a junção de um elemento a uma lista.

Sendo assim a definição de uma função cujo domínio de definição seja o tipo lista terá de explorar esses dois casos, definindo desse modo duas equações, uma para cada um dos padrões possíveis. Uma para o padrão `[]`,

```
procura1 (elem,[]) = False
```

e outra para o padrão `:`,

```
procura1 (elem,hd:tl) =∘ ..
```

É importante notar que as equações (e por isso os padrões) são lidos sequencialmente e como tal o primeiro "encaixe" define qual das equações é que se deve aplicar.

Por exemplo, qual das definições seguintes é a correcta, e porquê?

```
fact 0 = 1
fact n = n*fact(n-1)
```

ou

```
fact n = n*fact(n-1)
fact 0 = 1
```

Como fica claro no exemplo anterior os padrões possíveis também se aplicam aos construtores 0-ários, ou seja aos valores concretos de um dado tipo.

Em alguns casos o valor de um padrão, ou de parte dele, não é importante para a definição da função, nesses casos é possível definir o *padrão livre* "_" ("wildcard pattern"). Por exemplo na definição da função que serve para determinar se uma dada lista está vazia ou não.

```
estavazia :: [a] → Bool
estavazia [] = True
estavazia _  = False
```

Para o caso em que a lista não é vazia, isto é não houve encaixe com o padrão `[]`, os valores da cabeça e da cauda da lista não são importantes, então basta colocar o padrão livre para completar a definição. O padrão livre encaixa com qualquer valor, como tal terá de ser sempre o último caso a ser considerado.

3.5.1 A construção `case`

A utilização da técnica de associação de padrões pode ser alargada a outros valores que não os argumentos das funções. Suponhamos que, dado uma lista de expressões numéricas simples dadas sob a forma de uma sequência de caracteres contendo sempre um só sinal de operação, queremos calcular cada um dos valores associados às diferentes expressões contidas na lista obtendo deste modo a lista de todos os resultados.

Uma forma elegante de resolver este problema seria o seguinte:

```
calc :: [String] → [Int]
calc [] = []
calc (hd:tl) =
    case operador(hd) of
             '+' → (operando1(hd) + operando2(hd)):calc tl
             '*' → (operando1(hd) * operando2(hd)):calc tl
             '-' → (operando1(hd) - operando2(hd)):calc tl
             '/' → (operando1(hd) `div` operando2(hd)):calc tl
             _   → error "Operador não considerado"
```

com `operador`, `operando1`, e `operando2` funções de reconhecimento para as várias componentes das expressões simples.

Como vemos no exemplo a associação de padrões é, neste caso, feita sobre o resultado de uma expressão e não directamente sobre os argumentos da função.

A forma genérica da construção `case` é a seguinte:

```
case <e> of
  <p1> → <e1>
  <p2> → <e2>
  λcdots
  <pn> → <en>
```

com os e_is expressões e os p_is padrões.

3.6 Definições Locais

Em muitas situações o cálculo do valor de uma função depende de um determinado cálculo auxiliar. Embora se possa desenvolver esse cálculo auxiliar de uma forma independente, e depois usá-lo aquando da avaliação da função essa não é, em geral, a solução pretendida dado a utilidade do cálculo auxiliar se esgotar no cálculo do valor da função. Por exemplo na definição da função que calcula as raízes reais de uma equação do segundo grau.

```
raizes_reais :: (Float,Float,Float) → (Float,Float)
raizes_reais(a,b,c) = if b^2-4*a*c≥0 then
     ((-b+sqrt(b^2-4*a*c))/(2*a),(-b-sqrt(b^2-4*a*c))/(2*a))
                     else
     error "Raízes Imaginárias"
```

embora correcto esta definição é "desajeitada"; o cálculo do binómio discriminante é feito por três vezes; a definição da função resulta pouco clara.

A seguinte definição resolve esses problemas

```
raizes_reais1 :: (Float,Float,Float) → (Float,Float)
raizes_reais1 (a,b,c)
         | delta < 0  = error "Raízes Imaginárias"
         | delta ≥ 0 = (r1,r2)
         where d = 2*a
               delta = b^2-4*a*c
               r1 = (-b+sqrt(delta))/d
               r2 = (-b-sqrt(delta))/d
```

De uma forma geral tem-se

<*definição de função*> `where` <*definições locais*>

O contexto das definições locais em `where ...` é a equação em que está incluída.

As definições que se seguem ao identificador `where` não são visíveis no exterior da definição.

Este exemplo serve também para ilustrar a técnica da avaliação de expressões usada em Hashell, e que é designada por "lazy evaluation", avaliação preguiçosa numa tradução literal, e que eu vou traduzir por avaliação a pedido.

Vejamos em que consiste a avaliação a pedido, e em que isso interfere na definição que acabamos de escrever.

Numa linguagem em que a avaliação é forçada ("strict evaluation") o cálculo das expressões:

```
r1 = (-b+sqrt(delta))/d
r2 = (-b-sqrt(delta))/d
```

estaria errada para todos os valores de `a`, `b` e `c` que tornem `delta` negativo. Ou seja numa linguagem em que todas as componentes de uma expressão têm de ser avaliadas antes de ser calculado o valor final da função `raizes_reais1` está errada.

Em contrapartida em *Haskell* uma expressão só é avaliada quando é necessário. Ora o cálculo de `r1` (e de `r2`) só é necessário quando `delta >= 0`. Só nessa situação é que o valor de `r1` (e de `r2`) é calculado e como tal a definição de `raizes_reais1` está correcta.

Outra forma de escrever definições locais a uma função passa pela utilização do mecanismo `let ... in ...`. Podemos então re-escrever a definição da função `raizes_reais` da seguinte forma:

```
raizes_reais2 :: (Float,Float,Float) → (Float,Float)
raizes_reais2 (a,b,c)
            | b^2-4*a*c < 0 = error  error "Raízes Imaginárias"
            | b^2-4*a*c ≥ 0 = let
                               d = 2*a
                               delta = b^2-4*a*c
                               r1 = (-b+sqrt(delta))/d
                               r2 = (-b-sqrt(delta))/d
                              in (r1,r2)
```

De uma forma geral tem-se

`let` <*definições_locais*> `in` <*expressão*>

Temos então que o contexto das definições locais é dado pela expressão que se segue ao identificador `in`.

3.7 Funções de Ordem-Superior

Numa linguagem funcional as funções são objectos de primeira ordem, isto é, podem ser manipuladas tais como outros quaisquer objectos, nomeadamente podemos ter funções como argumentos de outras funções, assim como funções como resultado de outras funções. Um exemplo do que se acabou de dizer é-nos dado pela composição de funções, em *Haskell* podemos escrever `f.g` para deste modo obtermos a função composição de `f` com `g` (f após g).

Se num dos interpretadores de *Haskell* normalmente usados fizermos:

```
Main> :type (.)
```

obtemos como resposta o tipo da função composição:

```
forall b c a. (b → c) → (a → b) → a → c
```

ou seja, estamos perante uma função que recebe duas funções componíveis (vejam-se os tipos das funções) como argumento e que produz como resultado a função que é a composição das duas primeiras.

As funções que manipulam outras funções são designadas por funções de ordem-superior, ou mais usualmente, funcionais. Dentro destas é importante destacar algumas que pela sua utilidade são muito usadas em programação funcional, são elas:

map funcional que aplica uma dada função a todos os elementos de uma estrutura (em geral uma lista).

No Haskell temos a função `map` cujo tipo e definição são os seguintes:

```
map :: (a → b) → [a] → [b]
map f [] = []
map f (hd:tl) = f hd : map f tl
```

Como exemplo podemos ter uma função que adiciona 2 a todos os elementos de uma dada lista `mapm2 l = map (+2) l`

folding funcional que aplica de forma cumulativa uma dada função binária aos elementos de uma lista.

No Haskell temos a função `foldr1` cujo tipo e definição são os seguintes:

```
foldr1 ::  (a → a → a) → [a] → a
foldr1 f [a] = a
foldr1 f (a:b:tl) = f a (foldr1 f (b:tl))
```

Duas notas em relação a esta definição:

- pela definição da função `foldr1` fica claro que a as sucessivas aplicações da operação são feitas pela direita, o que implica um operador associativo à direita. Existe uma versão que faz a associação pela esquerda, é ela a função `foldl1`.
- a definição só está bem definida para listas não vazias. Se queremos considerar também este caso temos de dar mais um argumento, o qual especifica qual é o valor final no caso da lista ser vazia, as funções que implementam este caso são as funções `foldl` e `foldr` cujo tipo é: `forall a b.  (a -> b -> b) -> b -> [a] -> b`

Um exemplo de uma aplicação deste funcional é-nos dado pela implementação dos somatórios, `somatório l = foldr (+) 0 l`.

Filtros funcionais que aplicam um dado predicado a todos os elementos de uma estrutura (em geral uma lista) seleccionando deste modo todos os elementos que satisfazem um dado critério.

No Haskell temos a função `filter` cujo tipo e definição são os seguintes:

```
filter :: (a → Bool) → [a] → [a]
filter p [] = []
filter p (hd:tl)
    | p a = a : filter p tl
    | otherwise = filter p tl
```

Como exemplo de aplicação deste funcional temos a função `impares l = filter odd l` a qual nos dá todos os impares de uma dada lista de inteiros.

Capítulo 4

Definição de Tipos

Além dos tipos pré-definidos no `Prelude` é possível definir novos tipos. O *Haskell* permite construir novos tipos através das construções do produto, co-produto e exponenciação. Vejamos como proceder.

4.1 Definição por Enumeração (co-produto)

Através da declaração `data` podemos definir novos tipos. Uma das formas de o fazer é através da enumeração dos vários elementos que vão constituir o novo tipo. Por exemplo um tipo que permite a classificação dos triângulos através da relação entre os comprimentos das suas três arestas.

```
data Triangulo = NaoETriangulo | Escaleno | Isoscele | Equilatero
```

Estamos a definir um novo tipo, `Triângulo`, através da construção de um co-produto ("|"), definido pelos construtores 0-ários `NãoÉTriângulo`, ..., `Equilátero`. O novo tipo só vai ter cinco valores, quatro deles definidos pelos construtores, e $\perp$, valor que está presente em todos os tipos *Haskell*. O novo tipo não possui nenhuma operação.

Podemos agora definir uma função que, dados três inteiros x, y e z, dados por ordem crescente, classifique o triângulo formado pelas arestas de comprimento x, y e z.

```
classificar :: (Int,Int,Int) → Triangulo
classificar (x,y,z)
   | x+y < z = NaoETriangulo
   | x == z  = Equilatero
   | (x == y) || (y == z) = Isoscele
   | otherwise = Escaleno
```

De notar a utilização do valor lógico `otherwise` ($\doteq$ `True`) como forma de simplificar a escrita do último caso. No entanto se se tentar utilizar esta função vamos obter o seguinte resultado.

```
Main> classificar(1,2,3)
ERROR: Cannot find "show" function for:
*** expression : classificar (1,2,3)
*** of type    : Triangulo
```

Traduzindo: "não consigo achar uma função para visualizar os valores do tipo Triângulo". Ou seja em *Haskell* é necessário definir uma função de visualização para cada um dos seus tipos, ao definir-se um novo tipos essa função está em falha, veremos de seguida como proceder.

4.2 Classes

Em *Haskell* é possível definir novos tipos e inseri-los na estrutura de classe de tipos. Desta forma é possível obter funções para comparar valores dos novos tipos, funções para visualizar os novos valores, entre outras.

A estrutura de classe do *Haskell* dá conta das propriedades comuns aos diferentes tipos; por exemplo se os tipos admitem a comparação entre os seus membros, se admitem uma ordenação dos seus membros, entre outras.

Na figura 4.1 apresenta-se a estrutura de classes do *Haskell*.

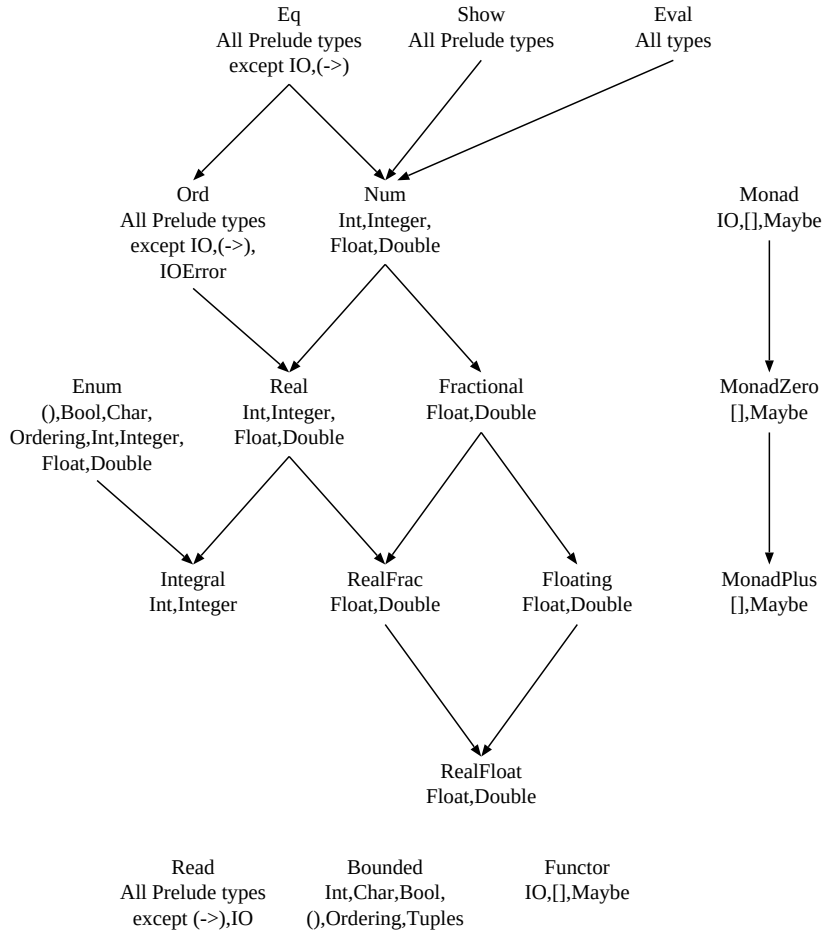

Figura 4.1: Classes Haskell

Tomando o exemplo do tipo `Triângulo`, vejamos como incorporar este novo tipo na estrutura já existente.

Pretende-se ter acesso às funções de visualização e de comparação.

A classe dos tipos para os quais é possível testar a igualdade entre os seus elementos é a classe `Eq`. Para incorporar o novo tipo nesta classe é necessário criar uma nova instância da classe e definir explicitamente o operador de igualdade para os elementos do novo tipo.

```
instance Eq Triangulo where
     NaoETriangulo == NaoETriangulo = True
     NaoETriangulo == Escaleno = False
```

```
NaoETriangulo == Isoscele = False
NaoETriangulo == Equilatero = False
Escaleno == NaoETriangulo = False
Escaleno == Escaleno = True
Escaleno == Isoscele = False
Escaleno == Equilatero = False
Isoscele == NaoETriangulo = False
Isoscele == Escaleno = False
Isoscele == Isoscele = True
Isoscele == Equilatero = False
Equilatero == NaoETriangulo = False
Equilatero == Escaleno = False
Equilatero == Isoscele = False
Equilatero == Equilatero = True
```

A definição da relação `=/=`, não é necessário dado que esta é definida na classe `Eq` à custa da relação `==`. A definição da relação de igualdade desta forma é fastidiosa e, para tipos com mais elementos, rapidamente se torna impraticável.

Podemos usar a estrutura de classes e os operadores definidos nas diferentes classes como forma de obter uma definição mais compacta. Para tal vamos começar por estabelecer o tipo `Triângulo` como uma instância da classe `Enum`, a classe dos tipos enumeráveis.

```
instance Enum Triangulo where
  fromEnum NaoETriangulo = 0
  fromEnum Escaleno = 1
  fromEnum Isoscele = 2
  fromEnum Equilatero = 3
  toEnum 0 = NaoETriangulo
  toEnum 1 = Escaleno
  toEnum 2 = Isoscele
  toEnum 3 = Equilatero
```

Podemos agora aproveitar o facto de o tipo `Int` ser uma instância das classes `Eq` e `Ord` (entre outras), e as definições de `fromEnum` e de `toEnum`, para definir o tipo `Triângulo` como uma instância das classe `Eq` e `Ord` de uma forma simplificada.

```
instance Eq Triangulo where
  (x == y) = (fromEnum x == fromEnum y)

instance Ord Triangulo where
  (x < y) = (fromEnum x < fromEnum y)
```

No entanto, e dado que todas estas definições são óbvias, seria de esperar que o *Haskell* providenciasse uma forma automática de estabelecer estas instância de um tipo definido por enumeração. Felizmente tal é possível através da seguinte definição definição do tipo `Triângulo`.

```
data Triangulo = NaoETriangulo | Escaleno | Isoscele | Equilatero
  deriving (Eq,Enum,Ord,Show)
```

Com esta definição são criadas, automaticamente, instâncias do novo tipo nas classes `Eq`, `Enum`, `Ord` e `Show`.

Podemos então avaliar as seguintes expressões:

```
Main> classificar(4,4,4)
```

```
Equilatero
Main> classificar(4,4,4) == classificar(2,2,5)
False
```

4.2.1 Classes Básicas

Vejamos de seguida, mais em pormenor, as classes de base da estrutura de classes do *Haskell*.

Tipos que Admitem Igualdade `Eq`

A classe `Eq` contém todos os tipos cujos elementos podem ser comparados em termos da sua igualdade. A classe possuí os dois métodos seguintes:

```
(==) :: a → a → Bool
```

É de notar que a classe das funções não é uma instância desta classe dado não ser, em geral, possível comparar duas funções em termos da sua igualdade.

Tipos Ordenados

A classe `Ord` é a classe que contém todos os tipos sobre os quais é possível definir uma função de ordenação total. Os tipos nesta classe admitem as funções de comparação usuais:

```
(<) :: a → a → Bool
(≤) :: a → a → Bool
(>) :: a → a → Bool
(≥) :: a → a → Bool
min :: a → a → a
max :: a → a → a
```

Todos os tipos básicos `Bool`, `Char`, `String`, `Int`, `Integer`, e `Float` são instâncias da classe `Ord`, assim como as listas e os tuplos, sempre que os seus elementos constituintes estejam em classes que sejam instâncias desta classe.

Tipos Visualizáveis e Passíveis de Leitura

A classe `Show` contém todos os tipos para as quais é possível converter os seus elementos numa dada "string" para posterior visualização, ou impressão.

Esta classe contém o seguinte método:

```
show :: a → String
```

A classe `Read` é dual da classe `Show` no sentido que contém todos os tipos para os quais é possível converter uma dada "string" num elemento válido do tipo. A classe possuí o método:

```
read :: String → a
```

Todos os tipos básicos `Bool`, `Char`, `String`, `Int`, `Integer`, e `Float` são instâncias das classes `Show` e `Read`, assim como as listas e os tuplos, sempre que os seus elementos constituintes estejam em classes que sejam instâncias destas duas classes.

Tipos Numéricos

As classes `Num`, `Integral` e `Fraccional` contém todos os tipos numéricos. A classe `Num` contém os métodos:

```
(+) :: a → a → → a
(-) :: a → a → → a
(*) :: a → a → → a
negate :: a → a
abs :: a → a
signum :: a → a
```

O método `negate` devolve o valor negativo de um dado número, o método `abs`, o seu valor absoluto, e `signum` o seu sinal: 1 para positivo, -1 para negativo.

A operação de divisão não está presente dado ser necessário distinguir entre divisão de números inteiros e divisão de números reais, as quais vão ser implementadas em duas classes distintas.

A classe `Integral` contém todos os métodos da classe `Num` (ver Figura 4.1), definindo ainda os seguintes métodos:

```
div :: a → a → a
mod :: a → a → a
```

os quais definem as operações usuais de divisão inteira e do resto da divisão inteira.

A classe `Fraccional`, está numa posição distinta da classe `Integral` no grafo das classes Haskell (Figura 4.1) estando no entanto abaixo da classe `Num` herdando desta todos os seus métodos e extendendo-a com os seguintes métodos:

```
(/) :: a → a → a
recip :: a → a
```

O tipo `Float` é uma instância da classe `Fraccional`. O método `/` define a operação de divisão usual e o método `recip` define o recíproco de um número fraccional.

4.3 Definição Paramétrica

A definição de novos tipos passa não só pela construção de co-produtos com construtores 0-ários, mas também pela construção de co-produtos, produtos e exponenciações com construtores n-ários. Por exemplo:

```
data Complexos = Par (Float,Float)
  deriving (Eq,Show)
```

define o tipo `Complexo` com um construtor binário que constrói um complexo como um par de reais.

De uma forma mais geral podemos definir tipos paramétricos, isto é tipos em cuja definição entram variáveis de tipo, e que desta forma não definem um só tipo mas um "classe" de tipos. Por exemplo, o tipo `Pares` pode ser definido do seguinte modo:

```
data Pares a b = ConstPares (a,b)
 deriving (Eq,Show)
```

com a definição das funções de projecção a serem definidas através da associação de padrões dados pelo construtor de pares `ConstPares`.

```
projX :: Pares a b → a
projX (ConstPares (x,y)) = x

projY :: Pares a b → b
projY (ConstPares (x,y)) = y
```

4.4 Definição Recursiva

É possível estender as definições de tipos ao caso das definições recursivas. Por exemplo o tipo `Lista` pode ser definido do seguinte modo:

```
data Lista a = Vazia | Lst (a,Lista a)
    deriving (Eq,Show)
```

Nesta caso temos dois construtores: `Vazia`, um construtor 0-ário, e `Lst` um construtor binário.

As definições de funções sobre elementos deste novo tipo podem ser definidas através de associação de padrões, tendo em conta os dois construtores disponibilizados.

Por exemplo a função que calcula o comprimento de uma lista.

```
comprimento :: (Lista a) → Int
comprimento Vazia = 0
comprimento (Lst(x,l)) = 1+comprimento(l)
```

como exemplo de aplicação desta função temos:

```
Main> comprimento(Lst(1,Lst(2,Lst(3,Vazia))))
3
```

Outro exemplo de uma definição paramétrica e recursiva é nos dada pela definição de árvores binárias.

```
data ArvBin a = Folha a | Ramo (ArvBin a,ArvBin a)
 deriving (Show)
```

A definição da função de travessia `inordem` é nos dada por:

```
inordem :: (ArvBin a) → [a]
inordem (Folha x) = [x]
inordem (Ramo(ab1,ab2)) = (inordem ab1 ) ++ (inordem ab2)
```

um exemplo de utilização é o seguinte:

```
Main> inordem(Ramo(Ramo(Folha 1,Folha 4),Folha 3))
[1, 4, 3]
```

Capítulo 5

Leitura e Escrita

A leitura e a escrita em *Haskell* é feita de uma forma puramente funcional através da Monade `IO`. Não vou aqui descrever as monades nem a forma como o *Haskell* formaliza as entradas e saídas à custa das monades, vou limitar-me a descrever as funções de leitura e escrita presentes no `Prelude`.

5.1 Funções de Escrita

As funções de escrita vão ter todas co-domínio na monade `IO`. Temos assim as seguintes funções de escrita de valores.

`putChar :: Char -> IO ()`	Escreve um caracter
`putStr :: String -> IO ()`	Escreve uma sequência de caracteres
`putStrLn :: String -> IO ()`	Escreve uma sequência de caracteres e muda de linha
`print :: Show a => a -> IO ()`	Escreve um valor.

5.2 Funções de Leitura

As funções leitura vão ter diferentes co-domínios conforme o seu objectivo.

`getChar :: IO Char`	Lê um caracter
`getLine :: IO String`	Lê uma linha e converte-a numa só sequência de caracteres
`getContents :: IO String`	Lê todo o conteúdo da entrada e converte-a numa só sequência de caracteres
`interact :: (String -> String) -> IO ()`	recebe uma função de sequências de caracteres para sequências de caracteres como argumento. Todo o conteúdo da entrada é passado como argumento para essa função, e o resultado dessa aplicação é visualizado.
`readIO :: Read a => String -> IO a`	Lê uma sequência de caracteres.
`readLine :: Read a -> IO a`	Lê uma sequência de caracteres e muda de linha.

5.3 Sequenciação

A forma de combinar vários comandos de escrita (ou de leitura) é feita no âmbito dos operadores monádicos. Temos então dois operadores que nos permitem combinar várias instruções de entrada/saída.

`>>`	Combina dois comandos sequencialmente.
`>>=`	Combina dois comandos sequencialmente, passando o resultado do primeiro como argumento para o segundo.

5.3.1 A notação `do`

Uma sequência ordenada de instruções pode ser escrita de uma forma simplificada através da notação `do`. Temos então:

```
do <padrão associado ao resultado de uma acção> <- <acção>
   <definições locais>
```

Por exemplo:

```
le :: IO()
le = do c <- getChar
        putChar c
```

5.4 Um exemplo

Torres de Hanói Defina uma função que calcule, para um dado número de discos, a solução do problema das Torres de Hanói.

```
-- Move n discos da Torre A para a Torre C, com a Torre B como auxiliar
--
hanoi :: IO ()
hanoi = do putStr "Introduza um Inteiro (de 0 a 10)"
           c<-getChar
           putStrLn "\n\nOs movimentos necessários são:"
           hanoi1(fromEnum(c)-48)

hanoi1 :: Int -> IO()
hanoi1 n = putStr ("\n"++(move n 'A' 'C' 'B'))
          where
          move :: Int -> Char -> Char -> Char -> String
          move 1 orig dest aux =
              "Move um disco de "++[orig]++" para "++[dest]++"\n"
          move n orig dest aux =
              (move (n-1) orig aux dest)++
              (move 1 orig dest aux)++
              (move (n-1) aux dest orig)
```

Capítulo 6

Módulos em Haskell

Um módulo consiste num conjunto de definições (tipos e funções) com um interface claramente definido. O interface explícita o que é importado e/ou exportado por um dado módulo.

6.1 Definição de Módulos

Cabeçalho `module` <*Nome*> `where`
...

Importação `import` <*Nome*>
Através deste comando todas as definições existentes no módulo importado, e que são definidas neste como exportáveis, passam a ser locais ao presente módulo.

Exportação A definição das definições que um dado módulo exporta podem ser definidas das seguintes formas:

- por omissão todas definições do módulo.
- por declaração explícita no cabeçalho do módulo das definições que se pretende exportar. Por exemplo:

  `module` <*Nome*>`(`<*nome de funções e Tipos*> `where`
- No caso da definição dos tipos se se quiser que os construtores também sejam exportados é necessário explicita-los, por exemplo:

  `module` <*Nome*>`(`<... *Tipos(nome dos construtores)*...>`)` `where`

Quando um módulo é importado por outro podemos ainda explicitar o facto de que esse módulo é por sua vez exportado escrevendo.
`module` <*Nome*>`(`<...`module(`*nome*`)`...>`)` `where`

6.2 Um exemplo, o TAD Pilha

Vejamos um exemplo de definição e utilização dos módulos através da implementação do *TAD Pilha*.

Implementação com Exportação Implícita: vejamos uma primeira implementação em que, por omissão da nossa parte, todos os elementos do módulo são exportados.

```
module Pilha where
data Pilha a = Pvazia | Pl (a,Pilha a)
                  deriving (Show)

top :: Pilha a -> a
top Pvazia = error "Top - Pilha vazia"
top (Pl(a,p)) = a

pop :: Pilha a -> Pilha a
pop Pvazia = error "Pop - Pilha vazia"
pop (Pl(a,p)) = p

push :: (a,Pilha a) -> Pilha a
push (a,p) = Pl(a,p)

évazia :: Pilha a -> Bool
évazia Pvazia = True
évazia _ = False

vazia :: () -> Pilha a
vazia() = Pvazia
```

Como podemos ver é a definição usual do *TAD Pilha* encapsulado num módulo *Haskell*, como nada foi dito no cabeçalho toda a informação é exportada.

Vejamos agora uma segunda implementação com a definição explícita do que se pretende exportar.

Implementação com Exportação Explícita: para exportar só aquilo que deve ser vísivel fora do módulo é necessário explicitar toda a informação que será vísivel no exterior.

```
module Pilha(Pilha,vazia,pop,top,push,évazia) where
data Pilha a = Pvazia | Pl (a,Pilha a)
                  deriving (Show)

top :: Pilha a -> a
top Pvazia = error "pilha vazia"
top (Pl(a,p)) = a

pop :: Pilha a -> Pilha a
pop Pvazia = error "pilha vazia"
pop (Pl(a,p)) = p

push :: (a,Pilha a) -> Pilha a
push (a,p) = Pl(a,p)
```

```
évazia :: Pilha a -> Bool
évazia Pvazia = True
évazia _ = False

vazia :: () -> Pilha a
vazia() = Pvazia
```

É de notar que, aqui tudo o que se omite não é exportado, nomeadamente como não declaramos no cabeçalho os construtores do novo tipo estes não será vísiveis do exterior, obtemos deste modo o comportamento desejado para um Tipo Abstracto de Dados.

Utilização: como exemplo de uma utilização do *TAD Pilha* temos o seguinte módulo em que se procede à linearização de uma pilha.

```
module UsaPilhas where

import Pilha

-- lineariza pilha
lineariza :: Pilha a -> [a]
lineariza p
    | évazia(p) = []
    | otherwise = top(p):lineariza(pop(p))
```

Toda a tentativa de utilização dos constructores provocará um erro dado que no módulo UsaPilhas eles não são conhecidos.

Referências

Bird, Richard. 1998. *Introduction to Functional Programming using Haskell*. Prentice Hall.

Hammond, Kevin, Peterson, John, Augustsson, Lennart, Fasel, Joseph, Gordon, Andrew, Peyton-Jones, Simon, & Reid, Alastair. 1997 (April 6). *Standard Libraries for the Haskell Programming Language*. Version 1.4 edn.

Hudak, Paul. 2000. *The Haskell School of Expression*. Cambridge University Press.

Hudak, Paul, Peterson, John, & Fasel, Joseph. 1997 (March). *A Gentle Introduction to Haskell, Version 1.4*. Tech. rept. Yale University and Univeristy of California.

Hutton, Graham. 2007. *Programming in Haskell*. Cambridge University Press.

Jones, Mark, & Peterson, John. 1997 (April). *Hugs 1.4*.

Peterson, John, Hammond, Kevin, Augustsson, Lennart, Boutel, Brian, Burton, Warren, Fasel, Joseph, Gordon, Andrew, Hughes, John, Hudak, Paul, Johnsson, Thomas, Jones, Mark, Meijer, Erik, Peyton-Jones, Simon, Reid, Alastair, & Philip, Wadler. 1997 (April, 7). *Report on the Programming Language Haskell*. Version 1.4 edn.

The GHC Team. *The Glasgow Haskell Compiler User's Guide*. `http://haskell.org/haskellwiki/GHC`.

Thompson, Simon. 1996. *Haskell, The Craft of Funcional Programming*. Harlow: Addison-Wesley.